Theoretical Computer Science 89 (1991) 137-159
Elsevier

Metacircularity in the polymorphic λ-calculus*

Frank Pfenning and Peter Lee

Department of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213-3890, USA

Abstract

Pfenning, F., and P. Lee, Metacircularity in the polymorphic λ-calculus, Theoretical Computer Science 89 (1991) 137-159.

We consider the question of whether a useful notion of metacircularity exists for the polymorphic λ-calculus. Even though complete metacircularity seems to be impossible, we obtain a close approximation to a metacircular interpreter. We begin by presenting an encoding for the Girard-Reynolds second-order polymorphic λ-calculus in the third-order polymorphic λ-calculus. The encoding makes use of representations in which abstractions are represented by abstractions, thus eliminating the need for the explicit representation of environments. We then extend this construction to encompass all of the ω-order polymorphic λ-calculus (F_ω). The representation has the property that evaluation is definable, and furthermore that only well-typed terms can be represented and thus type inference does not have to be explicitly defined. Unfortunately, this metacircularity result seems to fall short of providing a useful framework for typed metaprogramming. We speculate on the reasons for this failure and the prospects for overcoming it in the future. In addition, we briefly describe our efforts in designing a practical programming language based on F_ω.

1. Introduction

In this paper we consider the question of whether a useful notion of metacircularity exists for the polymorphic λ-calculus. There are, of course, many examples of metacircularity in untyped (or dynamically typed) languages, most notably in Lisp [20]. In [32], Reynolds gives a metacircular interpreter for a simple untyped functional language. This was pursued further by Steele and Sussman [35], and others. More recently, metacircularity has been explored for logic programming languages

* This research was supported in part by the Office of Naval Research under contract N00014-84-K-0145 and in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 5404, monitored by the Office of Naval Research under the same contract.

[1] and object-oriented languages [5]. In the realm of statically typed functional languages, however, we are unaware of any satisfactory examples. ML [21], for instance, seems not to be powerful enough to serve as its own metalanguage in a natural way—an ML interpreter written in ML would be highly redundant since, for example, type inference would have to be reimplemented explicitly. We would like to attain a high degree of *reflexivity*, meaning essentially that the metacircularity should be attained in a natural, "internal" way. We discuss this issue of reflexivity in greater detail in Section 2.

As a starting point for our investigation we chose the polymorphic λ-calculus. Conventional wisdom indicated that the answer to the question of whether metacircularity is possible in the polymorphic λ-calculus should be "No." It seemed that the type system would not permit a high degree of reflexivity, and also that, due to the strong normalization property of the calculus, the usual paradoxes would block our way.

We have found, however, that the answer is "Almost." After a brief review of the polymorphic λ-calculus in Section 3, we explain our answer in Section 4. We start by showing how the second-order polymorphic λ-calculus (F_2) (see [12, 13, 31]) can be represented in F_3 (the third-order polymorphic λ-calculus) extended with an ML-like facility for defining data types. This representation turns out to be inductively defined; hence we are able to define evaluation for the F_2 program representations via iteration. Then, by extending to higher orders the well-known methods for representing inductively defined data types in the second-order polymorphic λ-calculus [2, 31], we are able to present a complete encoding of F_2 programs in pure F_3 such that their evaluation function is definable in F_3.

Although quite different in the details, our construction is reminiscent of the *reflective tower* of Smith [33, 34]. Friedman and Wand's analysis of reflective towers [10, 36] emphasizes *reification*, the translation from programs to data, and *reflection*, the translation from data to programs, as central concepts. Thus, in the setting of a statically typed functional language, we have found elegant and concise definitions for limited forms of reification and reflection. This allows us to build a "tower," starting with an interpreter for F_2 written in F_3, and then extending to all higher orders by introducing a simple extension to the ω-order polymorphic λ-calculus (F_ω). This extension, which allows us to define reification and reflection functions for all of F_ω, is described in Section 5.

An interesting feature of our definitions is that only well-typed programs can be represented, or "reified." In the context of metaprogramming (i.e., the construction of programs that construct and manipulate other programs), this property implies that well-typed metaprograms can construct only well-typed object programs—a very desirable property. To our dismay, however, we have not been able to extend our metacircularity results to handle interesting kinds of metaprogramming problems. As part of our concluding remarks in Section 6, we speculate briefly on the reasons for this failure, as well as the prospects for overcoming this in the future. In addition, our experience has led us to ask whether F_ω can be used as the core

of a practical programming language. Here we find the situation to be much more encouraging, and so we also briefly describe our efforts to design a language based on F_ω, which we have called LEAP.

2. Reflexivity

We are concerned not only with metacircularity, but also with how easily and naturally the metacircularity can be expressed. We call this the *reflexivity* of the language. We will not attempt to give a formal definition for when a language is reflexive. Instead, we will try to give some informal criteria for judging the degree of reflexivity of a language, the basic one being the ability of a language to serve as its own metalanguage. This by itself does not seem enough, since then every Turing-complete language would be reflexive. In addition, we would like to require that the metacircularity is achieved in a natural, "internal" way. The answers to the following questions provide some hints for evaluating the degree of reflexivity of a language.

- How redundant is the definition of a metacircular interpreter? In a highly reflexive language, the metacircular interpreter should be simple and direct. The more that features of the object language can be implemented by using the corresponding features of the metalanguage, the more reflexive the language. We call this phenomenon *inheritance* of object language features from the metalanguage. Typical examples of features for which inheritance might be desirable are evaluation order (e.g., call-by-value vs. call-by-name) and static typechecking.
- How much of the metalanguage can be interpreted by the metacircular interpreter? Ideally, the metalanguage and object language should coincide.
- Can we define the functions `reify` and `reflect` in addition to `eval`? That is, can we coerce data into programs and vice versa?
- How well can object language syntax and metalanguage syntax be integrated? We will mainly ignore this issue: with the aid of good syntax-handling tools one should always be able to achieve a reasonably smooth integration of metalanguage and object language.

The concept of *inheritance* (though not under this name) was already considered by Reynolds in [32]. As we mentioned before, an ML interpreter written in ML would likely be highly redundant, since type inference would have to be reimplemented explicitly. In other words, it seems that ML type inference cannot be inherited, in part because of the complexity of the data type of programs, and also because of the implicit nature of the type quantification in ML. An interpreter written for a dynamically scoped Lisp would also be redundant, since environments must be represented and manipulated explicitly by the interpreter. The notion of variable binding cannot be inherited and must be programmed explicitly. However, many other features such as automatic storage management clearly are inherited in a typical metacircular Lisp interpreter.

In this paper we show how a high degree of reflexivity can be obtained in the polymorphic λ-calculus, despite the complications of types and the strong normalization property.

3. The ω-order polymorphic λ-calculus

In [12, 13], Girard defined a powerful extension to Church's simply typed λ-calculus [3] and goes on to give a constructive proof of strong normalization for his system. A fragment of Girard's calculus was independently discovered by Reynolds [31] who introduced abstraction on type variables and application of functions to types in order to define polymorphic functions explicitly. Reynolds' calculus is known as the second-order polymorphic λ-calculus.

Here we consider the ω-order polymorphic λ-calculus (F_ω) which is an extension of Reynolds' system but only a fragment of Girard's system (since it omits existentially quantified types). Our presentation of the calculus contains four distinct syntactic categories: *kinds*, *types*, *terms*, and *contexts*.

Going beyond the second-order polymorphic λ-calculus means that we have, in addition to types of terms, also functions from types to types, and so on. This generalization is essential for our construction, as even the representation of the simply typed λ-calculus appears to require the formation of functions from types to types. We refer collectively to types, functions from types to types, and so forth, as *higher-order types*, and the "functionality" of a higher-order type is referred to as its *kind*. Only higher-order types of kind "Type" can actually be the type of a term. These and other properties of the calculus are summarized at the end of this section. Throughout this paper, we will often say just "type" when we actually mean "higher-order type." In particular, we will refer to variables ranging over higher-order types simply as *type variables*.

We use the metavariables K, K' for kinds, $\alpha, \beta, \ldots$ for higher-order types and occasionally for type variables, θ for type variables, $M, N, \ldots$ for terms, and $x, y, \ldots$ for variables.

Definition 1. The syntactic categories of *kind*, *type*, *term*, and *context* are defined inductively by

Kinds	$K ::= \text{Type} \mid K \to K',$
(Higher-order) Types	$\alpha ::= \theta \mid \alpha \to \beta \mid \Delta\theta{:}K.\,\alpha \mid \lambda\theta{:}K.\,\alpha \mid \alpha\beta,$
Terms	$M ::= x \mid \lambda x{:}\alpha.\,M \mid MN \mid \Lambda\theta{:}K.\,M \mid M[\alpha],$
Contexts	$\Gamma ::= \langle\rangle \mid \Gamma, \theta{:}K \mid \Gamma, x{:}\alpha.$

The λ symbol is used to construct functions that can be applied to a term, yielding a term, and also to build functions that can be applied to a type, yielding a type. The symbol Λ constructs functions that can be applied to types, yielding a term.

Such a function will have a Δ type. The order of a term in this calculus is determined by what kind of abstractions over types are allowed: we obtain the second-order polymorphic λ-calculus (F_2) if we allow abstractions only over type variables of kind Type; we obtain F_3 with abstractions over type variables of kinds Type$\to\cdots\to$ Type, etc. Contexts uniquely assign kinds to type variables and types to term variables. We will omit empty contexts, and write "Γ, Γ'" for the concatenation of two contexts. It is convenient not to distinguish between variables in a global context and constants, and occasionally, in a slight abuse of language, we call a member of a context a *constant.*

Definition 2. The following judgments define the calculus F_ω:

$\vdash \Gamma$ context, Γ is a valid context,

$\vdash K \in \text{Kind}$, K is a valid kind,

$\Gamma \vdash \alpha \in K$, α has kind K in context Γ,

$\Gamma \vdash M \in \alpha$, M has type α in context Γ,

$\alpha =_{\beta\eta} \beta$.

We will regard α-convertible types and terms (with binders λ, Λ, and Δ) to be equal. Thus we will ignore the issues of variable renaming and name clashes. We also assume that any variable occurs at most once in the domain of a context Γ (that is, the list of all variables θ and x such that $\theta{:}K$ or $x{:}\alpha$ are in Γ). This can always be achieved by α-conversion or through the use of deBruijn indices, as for example in Coquand and Huet's presentation and implementation of the Calculus of Constructions [7, 9].

Definition 3 (*Valid kinds*).

$$\frac{}{\vdash \text{Type} \in \text{Kind}}$$

$$\frac{\vdash K \in \text{Kind} \qquad \vdash K' \in \text{Kind}}{\vdash K \to K' \in \text{Kind}}$$

Definition 4 (*Valid higher-order types*).

$$\frac{\Gamma \vdash \alpha \in \text{Type} \qquad \Gamma \vdash \beta \in \text{Type}}{\Gamma \vdash \alpha \to \beta \in \text{Type}}$$

$$\frac{\Gamma, \theta{:}K \vdash \alpha \in \text{Type}}{\Gamma \vdash \Delta\theta{:}K.\, \alpha \in \text{Type}}$$

$$\frac{\vdash \Gamma \text{ context} \qquad \theta{:}K \text{ in } \Gamma}{\Gamma \vdash \theta \in K}$$

$$\frac{\Gamma, \theta{:}K \vdash \alpha \in K'}{\Gamma \vdash \lambda\theta{:}K.\, \alpha \in K \to K'}$$

$$\frac{\Gamma \vdash \alpha \in K \to K' \qquad \Gamma \vdash \beta \in K}{\Gamma \vdash \alpha\beta \in K'}$$

Definition 5 (*Valid contexts*).

$$\frac{}{\vdash \langle\,\rangle \text{ context}}$$

$$\frac{\vdash \Gamma \text{ context} \qquad \vdash K \in \text{Kind}}{\vdash \Gamma, \theta{:}K \text{ context}}$$

$$\frac{\vdash \Gamma \text{ context} \qquad \Gamma \vdash \alpha \in \text{Type}}{\vdash \Gamma, x{:}\alpha \text{ context}}$$

Definition 6 (*Valid terms*).

$$\frac{\vdash \Gamma \text{ context} \qquad x{:}\alpha \text{ in } \Gamma}{\Gamma \vdash x \in \alpha}$$

$$\frac{\Gamma, x{:}\alpha \vdash M \in \beta}{\Gamma \vdash \lambda x{:}\alpha\,.\, M \in \alpha \to \beta}$$

$$\frac{\Gamma \vdash M \in \alpha \to \beta \qquad \Gamma \vdash N \in \alpha}{\Gamma \vdash M\,N \in \beta}$$

$$\frac{\Gamma, \theta{:}K \vdash M \in \beta}{\Gamma \vdash \Lambda\theta{:}K\,.\, M \in \Delta\theta{:}K\,.\,\beta}$$

$$\frac{\Gamma \vdash M \in \Delta\theta{:}K\,.\,\beta \qquad \Gamma \vdash \alpha \in K}{\Gamma \vdash M[\alpha] \in (\lambda\theta{:}K\,.\,\beta)\alpha}$$

$$\frac{\Gamma \vdash M \in \alpha \qquad \alpha =_{\beta\eta} \beta \qquad \Gamma \vdash \beta \in \text{Type}}{\Gamma \vdash M \in \beta}$$

The rule giving the type of a Λ-abstraction is formulated so that the type conversion rule can be used to carry out the substitution of α for the free occurrences of θ in β. The simple device of reducing substitution to β-reduction is used later for the representation of terms. When we omit a kind, as in "$\Delta\theta\,.\,\alpha$," we mean "$\Delta\theta{:}\text{Type}\,.\,\alpha$."

In the type conversion rule we allow conversion between $\beta\eta$-equivalent types. We define $=_{\beta\eta}$ on types as the equivalence relation induced by β- and η-reduction of types. At the level of types, a β-redex has the form $(\lambda\theta{:}K\,.\,\alpha)\gamma$ and an η-redex has the form $(\lambda\theta{:}K\,.\,\alpha\theta)$ where θ is not free in α.

In the conversions for terms we will have occasion to consider both β-conversion and η-conversion. Both must include type applications, that is, $(\Lambda\theta\,.\,M)[\beta] =_\beta [\beta/\theta]M$ (where $[\beta/\theta]M$ is the result of substituting β for θ in M, renaming bound type variables to avoid name clashes) and $\Lambda\theta\,.\,M[\theta] =_\eta M$ if θ is not free in M. We write $=_{\beta\eta}$ for the equivalence relation induced by this extended notion of conversion. A valid term M or type α is in *long $\beta\eta$-normal form* if it is in β-normal form and it cannot be η-expanded to a valid term or type without creating a β-redex.

During the remainder of the paper, we will make use of some fundamental properties of the calculus whose proofs can be found elsewhere (see, for example,

[12, 14, 11]) or follow immediately from known results. We state here only a few of them.

Theorem 7 (Basic properties of F_ω; Girard [14]).

(1) *If* $\Gamma \vdash M \in \alpha$ *then* $\Gamma \vdash \alpha \in \mathrm{Type}$.

(2) *If* $\Gamma \vdash \alpha \in K$ *then* α *has a unique long* $\beta\eta$*-normal form.*

(3) *If* $\Gamma \vdash M \in \alpha$ *then* M *has a unique* β*-normal form and a unique long* $\beta\eta$*-normal form.*

(4) *If* $\Gamma \vdash M \in \alpha$ *and* $\Gamma \vdash M \in \beta$ *then* $\alpha =_{\beta\eta} \beta$.

(5) $\Gamma \vdash M \in \alpha$ *is decidable.*

The system F_ω can be stratified into levels very naturally in analogy to the way higher-order logic can be stratified into first-order logic, second-order logic, and so on. Of course, this does not mean that our system is predicative: already the level F_2 is impredicative. In our setting the concept of order is determined exclusively by the kinds of the types of a term and its subterms. The orders are calibrated by naming the second-order polymorphic λ-calculus F_2 (the system Girard calls F). Since the notation of order is important in the discussion of metacircularity, we will give a formal definition of the overloaded function o which applies to kinds, types, terms, and contexts.

Definition 8 (*Order of kinds*). We define inductively:

$$o(\mathrm{Type}) = 1,$$

$$o(K \to K') = \max(o(K)+1, o(K')).$$

Definition 9 (*Order of types*). The order of all types are eventually reduced to the order of kinds. Given any context Γ, we define inductively o^Γ:

$$o^\Gamma(\theta) = o(K), \quad \text{where } \theta{:}K \text{ in } \Gamma,$$

$$o^\Gamma(\alpha \to \beta) = \max(o^\Gamma(\alpha), o^\Gamma(\beta)),$$

$$o^\Gamma(\Delta\theta{:}K\,.\,\alpha) = \max(o(K)+1, o^{\Gamma,\theta:K}(\alpha)),$$

$$o^\Gamma(\lambda\theta{:}K\,.\,\alpha) = \max(o(K)+1, o^{\Gamma,\theta:K}(\alpha)),$$

$$o^\Gamma(\alpha\beta) = \max(o^\Gamma(\alpha), o^\Gamma(\beta)).$$

Definition 10 (*Order of terms*). The order of all terms is eventually reduced to the order of types and kinds. We define inductively:

$$o^\Gamma(x) = o^\Gamma(\alpha), \quad \text{where } x{:}\alpha \text{ in } \Gamma,$$

$$o^\Gamma(\lambda x{:}\alpha\,.\,M) = \max(o^\Gamma(\alpha), o^{\Gamma,x:\alpha}(M)),$$

$$o^\Gamma(M\,N) = \max(o^\Gamma(M), o^\Gamma(N)),$$

$$o^\Gamma(\Lambda\theta{:}K\,.\,M) = \max(o(K)+1, o^{\Gamma,\theta:K}(M)),$$

$$o^\Gamma(M[\alpha]) = \max(o^\Gamma(\alpha)+1, o^\Gamma(M)).$$

Definition 11 (*Order of contexts*).

$$o(\Gamma, x{:}\alpha) = \max(o(\Gamma), o^{\Gamma}(\alpha)),$$
$$o(\Gamma, \theta{:}K) = \max(o(\Gamma), o(K)).$$

F_n is the restriction of F_ω to kinds, types, terms, and contexts of order n. According to this definition, F_1 will be the simply typed λ-calculus and F_2 is almost exactly Reynolds' second-order polymorphic λ-calculus. The difference is that F_2 as defined above allows explicit formation of functions from types to types with λ, which was not part of Reynolds' calculus. However, it can be shown that no $\alpha \in \text{Type}$ with $o(\alpha) = 2$ in normal form will contain such an abstraction, so the difference is minor (see [28] for a detailed discussion).

4. Reflection of F_2 in F_3

In this section we describe how metacircularity can be achieved to a large degree with F_ω. The presentation will proceed in two stages: first we show how the F_2 fragment can be represented in F_3 augmented by a few constants and some new reduction rules defining evaluation for F_2. Second we show how all these constants are actually definable in such a way that β-reduction is sufficient, that is, F_2 and its evaluation function can be faithfully represented in F_3.

4.1. Representation of programs

The first concern is the ability to represent programs in the language as data. Two approaches seem plausible: to build in a new special data type for programs, or to use combinations of existing built-in data types to represent programs. We first show how representation can be achieved using new constants, and then how these constants can be eliminated through internal definition.

Starting informally, it is useful to consider how programs in F_2 might be represented if ML-style datatype constructors were used. Looking at Definition 1 suggests that there ought to be five constructors: one for variables and one for each form of abstraction and application. Thus we make our first crucial decision: the *types* of F_2 are *not* represented explicitly, but rather mapped into types in F_3. This technique results in the property that only well-typed terms in F_2 can be represented, thereby providing a built-in type safety. But this then forces our hand in the representation of variables: if we try to represent them explicitly (say as strings or natural numbers), there appears to be no way to guarantee well-typedness of the term we are trying to represent.

The way out of this dilemma is to use the idea of *higher-order abstract syntax*, which goes back to Church and appears in different guises in various places in the literature, for example [4, 15, 25]. The essence of higher-order abstract syntax is to

use the abstraction mechanism of the metalanguage to implement abstraction in the object language. Here, of course, both metalanguage and object language will be fragments of F_ω, so λ and Λ will be used to implement themselves. That this is possible may seem unlikely at first, especially in this statically typed language, but, as we will see, we can construct such a representation and even define an evaluation function.

Ignoring the problems of types for the moment, we thus obtain the constructors `rep` (for bound variable occurrences), `lam`, `app`, `typlam`, and `typapp`. The constructors `lam` and `typlam` expect absstractions as arguments, since abstractions are to be represented by abstractions. In the interests of readability, we have in the definition below omitted the context argument of the representation function $\overline{(\,)}$ which could be easily filled in. The crucial property of this function is given in Theorem 15.

Definition 12 (*Standard representation*). Let M be a valid term of F_2 in some context Γ. We define the *standard representation* $\bar{M}$ of M inductively as follows:

If $x \in \alpha$	then $\bar{x} = \mathtt{rep}[\alpha]x$,
If $\lambda x{:}\alpha\,.\,M \in \alpha \to \beta$	then $\overline{\lambda x{:}\alpha\,.\,M} = \mathtt{lam}[\alpha][\beta](\lambda x{:}\alpha\,.\,\bar{M})$,
If $M \in \alpha \to \beta$ and $N \in \alpha$	then $\overline{M\,N} = \mathtt{app}[\alpha][\beta]\bar{M}\,\bar{N}$,
If $\Lambda\theta\,.\,M \in \Delta\theta\,.\,\alpha$	then $\overline{\Lambda\theta\,.\,M} = \mathtt{typlam}[\lambda\theta\,.\,\alpha](\Lambda\theta\,.\,\bar{M})$,
If $M \in \Delta\theta\,.\,\alpha$	then $\overline{M\,[\beta]} = \mathtt{typapp}[\lambda\theta\,.\,\alpha]\bar{M}[\beta]$.

Since the type of a given term is not unique (due to the type conversion rule), this does not actually define a function on terms. However, due to uniqueness of types up to conversion it is easy to see that $\bar{M}$ is also unique up to conversion at the level of types occurring in $\bar{M}$. We thus turn $\overline{(\,)}$ into a function by stipulating that the types in $\bar{M}$ be in normal form.

Example 13 (*Representation of the polymorphic identity function*). Let $\mathtt{id} = \Lambda\alpha\,.\,\lambda x{:}\alpha\,.\,x$. Then

$$\overline{\mathtt{id}} = \mathtt{typlam}[\lambda\theta\,.\,\theta \to \theta](\Lambda\alpha\,.\,\mathtt{lam}[\alpha][\alpha](\lambda x{:}\alpha\,.\,\mathtt{rep}[\alpha]x)).$$

What is the appropriate metalanguage in which to interpret $\bar{M}$? Obviously, we need five constructors and a representation type. We write "$\pi\,\alpha$" for the type of representations of programs of type α and define a context Π giving the types of the constructors.

Definition 14 (*Representation context* Π).

$$\begin{aligned}
\Pi = {} & \pi : \mathrm{Type} \to \mathrm{Type},\\
& \mathtt{rep} : \Delta\alpha\,.\,\alpha \to \pi\alpha,\\
& \mathtt{lam} : \Delta\alpha\,.\,\Delta\beta\,.\,(\alpha \to \pi\beta) \to \pi(\alpha \to \beta),\\
& \mathtt{app} : \Delta\alpha\,.\,\Delta\beta\,.\,\pi(\alpha \to \beta) \to \pi\alpha \to \pi\beta,\\
& \mathtt{typlam} : \Delta\alpha{:}\mathrm{Type} \to \mathrm{Type}\,.\,(\Delta\theta\,.\,\pi(\alpha\theta)) \to \pi(\Delta\theta\,.\,\alpha\theta),\\
& \mathtt{typapp} : \Delta\alpha{:}\mathrm{Type} \to \mathrm{Type}\,.\,\pi(\Delta\theta\,.\,\alpha\theta) \to \Delta\theta\,.\,\pi(\alpha\theta).
\end{aligned}$$

The significance of this definition lies in the following theorem.

Theorem 15 (Soundness of program representation). *Let M be a term such that $\Gamma, \Pi \vdash M \in \alpha$ where α is in normal form and $o^{\Gamma,\Pi}(M) = 2$. Then $\Gamma, \Pi \vdash \bar{M} \in \pi\alpha$. Moreover, $o^{\Gamma,\Pi}(\bar{M}) = 3$.*

Proof. By a simple induction on the derivation of $\Gamma, \Pi \vdash M \in \alpha$. □

Note also, that for a term M in F_1 (a simply-typed term), the representation $\bar{M}$ will be in F_2. For the representation of F_2-terms, it is not necessary to consider the context Π for M, but it is convenient to do so for the completeness theorem. For example, a term N such that $\Pi \vdash N \in \pi(\pi\alpha)$ does not directly represent a term in the empty context, though it represents a term in the context Π (see Example 17 and Theorem 18).

Definition 16 (*Representation*). We define the relation "*represents*" inductively like the standard representation, except that $\mathtt{rep}[\alpha]M$ (which is not the standard representation of any term unless M is a variable) is defined as representing M and if $N =_{\beta\eta} N'$ and N represents M, then N' also represents M.

Example 17 (*A representation of a representation of the polymorphic identity function*). Let $\overline{\mathtt{id}}$ be the standard representation of the polymorphic identity function from Example 13. The following represents $\overline{\mathtt{id}}$, though it is not its standard representation:

$$\mathtt{rep}[\pi(\Delta\alpha\,.\,\alpha \to \alpha)](\mathtt{typlam}[\lambda\theta\,.\,\theta \to \theta](\Lambda\alpha\,.\,\mathtt{lam}[\alpha][\alpha](\lambda x{:}\alpha\,.\,\mathtt{rep}[\alpha]x))).$$

Omitting some types within square brackets and without writing out representations of variables, the standard representation of $\overline{\mathtt{id}}$ is

$$\begin{aligned}&\mathtt{app}[\,][\,](\mathtt{typapp}[\,]\overline{\mathtt{typlam}}[\lambda\theta\,.\,\theta \to \theta])\\ &\quad(\mathtt{typlam}[\,](\Lambda\alpha\,.\,\mathtt{app}(\mathtt{typapp}[\,](\mathtt{typapp}[\,]\overline{\mathtt{lam}}[\alpha])[\alpha])\\ &\qquad\qquad(\mathtt{lam}[\,][\,](\lambda x{:}\alpha\,.\,\mathtt{app}[\,][\,](\mathtt{typapp}[\,]\overline{\mathtt{rep}[\alpha]})\bar{x})))).\end{aligned}$$

The type of both of these representations is $\pi(\pi(\Delta\alpha\,.\,\alpha \to \alpha))$ and they are valid terms in the context Π.

Theorem 18 (Completeness of program representation). *Let N be a term such that $\Gamma, \Pi \vdash N \in \pi\alpha$ for α in normal form and $o^{\Gamma,\Pi}(N) = 3$. Then there is an M with $\Gamma, \Pi \vdash M \in \alpha$ such that N represents M.*

Proof. We take the β-normal form of N and then η-expand to achieve the long $\beta\eta$-normal form N'. By definition, if N' represents M, then N also represents M. One can then see that N' must be of the form of a constructor (that is, a variable in Π) applied to sufficiently many arguments: none of the variables in Γ could produce a term of type $\pi\alpha$, since α is introduced in Π and can therefore not appear

in Γ. The function that maps N' back to the term that it represents is in essence the function `reflect` from Definition 20. □

Note that representations of programs are not unique, not even up to conversion. For example, any term $M \in \alpha$ in normal form can be represented as $\mathtt{rep}[\alpha]M$, but it also has a representation in terms of `lam`, `app`, `typlam`, `typapp`, and `rep`, where `rep` is applied only to variables.

Because of the property mentioned in the previous paragraph, it is tempting to try to eliminate the `rep` constructor from the representation. However, it is crucial in order to convert bound variables into their representations. A simple attempt to get around this would be to change the type of the `lam` constructor to $\Delta\alpha\Delta\beta.(\pi\alpha \to \pi\beta) \to \pi(\alpha \to \beta)$. However, this also fails since then there is a negative occurrence of π in the argument to one of the constructors for π, making it no longer inductive, but generally recursive. We have not explored in depth whether the addition of general recursive types would strengthen the possibility of reflection in F_ω, but we suspect that the results would be equally unsatisfactory.

4.2. Evaluation

Of course, there are a wide variety of possible representations, even given the constraint that we would like to represent only well-typed F_2 programs. One of the crucial properties of our representation is the definability of the evaluation function. What is meant by evaluation in the context of F_ω? Since the calculus has the strong normalization property with respect to both β and $\beta\eta$-reduction, the definition that appears to be easiest to work with is that M evaluates to N if N is in β-normal form and $M =_\beta N$.

The central idea in the definition of evaluation over the representation is to use a detour: we *reflect* the represented term into its corresponding term representation and then *reify* the result of the evaluation in the metalanguage. Let us try to explain this by analogy. Assume we have a programming language like ML (with a built-in type of integers) and we make an explicit definition of a data type of natural numbers (in a unary representation):

```
indtype nat: Type with
  zero: nat
  succ: nat → nat
  end
```

There is an obvious way of defining addition by using a schema of *iteration* (to be made precise below):

$$\mathtt{plus}\ \mathtt{zero}\ m = m$$
$$\mathtt{plus}\ (\mathtt{succ}\ n)m = \mathtt{succ}\ (\mathtt{plus}\ n\ m)$$

It turns out that in our setting such a straightforward definition of evaluation is not possible, but there is a more devious definition of addition whose idea carries over to our example. We define representation and "unrepresentation" functions,

let us call them reflectnat and reifynat.

$$\begin{aligned}&\texttt{reflectnat} : \textbf{nat} \to \textbf{int}\\ &\texttt{reflectnatzero} = 0\\ &\texttt{reflectnat}\ (\texttt{succ}\ n) = (\texttt{reflectnat}\ n) + 1\\ &\\ &\texttt{reifynat} : \textbf{int} \to \textbf{nat}\\ &\texttt{reifynat}\ n = \textbf{if}\ n = 0\ \textbf{then}\ \texttt{zero}\ \textbf{else}\ \texttt{succ}\ (\texttt{reifynat}\ (n-1))\end{aligned}$$

Then addition can be programmed by observing that + on integers "behaves like" plus on natural numbers, that is,

$$\texttt{reflectnat}\ n + \texttt{reflectnat}\ m = \texttt{reflectnat}\ (\texttt{plus}\ n\ m).$$

Given that we have a reification function we can then define

$$\texttt{plus}\ n\ m = \texttt{reifynat}\ (\texttt{reflectnat}\ n + \texttt{reflectnat}\ m).$$

Our construction for programs follows this development, with functions reflect (for reflectnat) and reify (for reifynat) and eval (for plus). What plays the role of +? In essence, application does, since evaluating a function application in the metalanguage models the evaluation of a term (which is not in normal form) in the object language.

Before we can give the definition of an evaluation function, we have to be more precise about the tools that will let us define functions over constructors as done informally above. All that we need here is the schema of *iteration over an inductively defined type.* For a general development of the notions of inductively defined types and iteration over such types that is general enough to apply to the representation of terms in F_2, see [8, 27]; here we only sketch some of the essential elements. An inductively defined type is given by a list of its constructors and their types. This is an extension of the **datatype** construction in ML, since constructors may be explicitly polymorphic. It is shown in [27] (extending ideas of Böhm and Berarducci [2]) that these types do not require an addition to the core language, since inductively defined types are *representable* by closed types (see Section 4.4). With this in mind, we can now present a specification of the type of programs:

$$\begin{aligned}&\textbf{indtype}\ \pi : \mathrm{Type} \to \mathrm{Type}\ \textbf{with}\\ &\quad \mathrm{rep} : \Delta\alpha\,.\,\alpha \to \pi\alpha\\ &\quad \mathrm{lam} : \Delta\alpha\,.\,\Delta\beta\,.\,(\alpha \to \pi\beta) \to \pi(\alpha \to \beta)\\ &\quad \mathrm{app} : \Delta\alpha\,.\,\Delta\beta\,.\,\pi(\alpha \to \beta) \to \pi\alpha \to \pi\beta\\ &\quad \mathrm{typlam} : \Delta\alpha{:}\mathrm{Type} \to \mathrm{Type}\,.\,(\Delta\theta\,.\,\pi(\alpha\theta)) \to \pi(\Delta\theta\,.\,\alpha\theta)\\ &\quad \mathrm{typapp} : \Delta\alpha{:}\mathrm{Type} \to \mathrm{Type}\,.\,\pi(\Delta\theta\,.\,\alpha\theta) \to \Delta\theta\,.\,\pi(\alpha\theta)\\ &\quad \textbf{end}\end{aligned}$$

It is important to note that this is indeed inductive, that is, all occurrences of π in the types of the arguments of the constructors are positive. This allows the definition of a function over the inductive type by *iteration*, a simpler form of primitive recursion. The general schema, instantiated to the type π, yields the following.

Definition 19 (*Iteration over* π). Given Θ:Type → Type and terms

$$h_1 \in \Delta\alpha\,.\,\alpha \to \Theta\alpha,$$
$$h_2 \in \Delta\alpha\,.\,\Delta\beta{:}\mathrm{Type}\,.\,(\alpha \to \Theta\beta) \to \Theta(\alpha \to \beta),$$
$$h_3 \in \Delta\alpha\,.\,\Delta\beta\,.\,\Theta(\alpha \to \beta) \to \Theta\alpha \to \Theta\beta,$$
$$h_4 \in \Delta\alpha{:}\mathrm{Type} \to \mathrm{Type}\,.\,(\Delta\theta\,.\,\Theta(\alpha\theta)) \to \Theta(\Delta\theta\,.\,\alpha\theta),$$
$$h_5 \in \Delta\alpha{:}\mathrm{Type} \to \mathrm{Type}\,.\,\Theta(\Delta\theta\,.\,\alpha\theta) \to \Delta\theta\,.\,\Theta(\alpha\theta).$$

If f satisfies

$$f[\alpha](\mathtt{rep}[\alpha]x) = h_1[\alpha]x,$$
$$f[\alpha \to \beta](\mathtt{lam}[\alpha][\beta]x) = h_2[\alpha][\beta](\lambda y{:}\alpha\,.\,f[\beta](xy)),$$
$$f[\beta](\mathtt{app}[\alpha][\beta]xy) = h_3[\alpha][\beta](f[\alpha \to \beta]x)(f[\alpha]y),$$
$$f[\Delta\theta\,.\,\alpha\theta](\mathtt{typlam}[\alpha]x = h_4[\alpha](\Lambda\theta\,.\,f[\alpha\theta](x[\theta])),$$
$$f[\alpha\beta](\mathtt{typapp}[\alpha]x[\beta]) = h_5[\alpha]f[\Delta\theta\,.\,\alpha\theta]x[\beta],$$

then $f{:}\Delta\alpha{:}\mathrm{Type}\,.\,\pi\alpha \to \Theta\alpha$ is defined from $h_1, \ldots, h_5$ by iteration over π at type Θ.

Given this general schema it is easy to define the reflection function.

Definition 20 (*Function* `reflect`).

$$\mathtt{reflect} : \Delta\alpha\,.\,\pi\alpha \to \alpha,$$
$$\mathtt{reflect}[\alpha](\mathtt{rep}[\alpha]x) = x,$$
$$\mathtt{reflect}[\alpha \to \beta](\mathtt{lam}[\alpha][\beta]x) = \lambda y{:}\alpha\,.\,\mathtt{reflect}[\beta](xy),$$
$$\mathtt{reflect}[\beta](\mathtt{app}[\alpha][\beta]xy) = (\mathtt{reflect}[\alpha \to \beta]x)(\mathtt{reflect}[\alpha]y),$$
$$\mathtt{reflect}[\Delta\theta\,.\,\alpha\theta](\mathtt{typlam}[\alpha]x) = \Lambda\theta\,.\,\mathtt{reflect}[\alpha\theta](x[\theta]),$$
$$\mathtt{reflect}[\alpha\beta](\mathtt{typapp}[\alpha]x[\beta]) = \mathtt{reflect}[\Delta\theta\,.\,\alpha\theta]x[\beta].$$

It is easy to verify that this is an instance of the schema for iteration given above where $\Theta = \lambda\theta\,.\,\theta$. The crucial property is that this really defines a proper reflection function with respect to the reification function $\overline{(\,)}$ (see Theorem 22). In order to properly formulate this theorem, we have to add another constant to the context Π and some new conversion rules. In the end this will turn out to be unnecessary, since we can find a way of repressenting inductively defined types in the pure calculus (with an empty context) in such a way that iteration is definable.

Definition 21 (*Iteration context* Π^+). We add a variable to the representation context Π that expects a Θ and then functions $h_1, \ldots, h_5$ to return the function that is defined by iteration at type Θ from $h_1, \ldots, h_5$.

$$\begin{aligned}\Pi^+ = \Pi, \mathtt{itprog} : \Delta\Theta{:}\mathrm{Type} \to \mathrm{Type}\,.\\ &\to \Delta\alpha\,.\,\alpha \to \Theta\alpha\\ &\to \Delta\alpha\,.\,\Delta\beta\,.\,(\alpha \to \Theta\beta) \to \Theta(\alpha \to \beta)\\ &\to \Delta\alpha\,.\,\Delta\beta\,.\,\Theta(\alpha \to \beta) \to \Theta\alpha \to \Theta\beta\\ &\to \Delta\alpha{:}\mathrm{Type} \to \mathrm{Type}\,.\,(\Delta\theta\,.\,\Theta(\alpha\theta)) \to \Theta(\Delta\theta\,.\,\alpha\theta))\\ &\to \Delta\alpha{:}\mathrm{Type} \to \mathrm{Type}\,.\,\Theta(\Delta\theta{:}\mathrm{Type}\,.\,\alpha\theta) \to \Delta\theta\,.\,\Theta(\alpha\theta)\\ &\to \Delta\alpha\,.\,\pi\alpha \to \Theta\alpha.\end{aligned}$$

The *iterative reduction property* of itprog states that $f = \texttt{itprog}\Theta h_1 \ldots h_5$ satisfies the equations from Definition 19 (as reductions, they would be read from left to right). The corresponding enriched equivalence relation is denoted by $=_{\beta\eta\iota}$.

Theorem 22 (Correctness of reflect). *Let* $\Gamma, \Pi \vdash N \in \pi\alpha$ *be some (not necessarily standard) representation of the term* M. *Then* $\texttt{reflect}\ N =_{\beta\eta\iota} M$.

Proof. As in the proof of Theorem 18 by induction on the long normal form of N in terms of the constructors of π. □

4.3. *The definitions of* reify *and* eval

Given the definition of reflect, it is a simple matter to give the definition of $\texttt{eval} : \pi\alpha \to \pi\alpha$. Intuitively, eval should take the representation of a term and return a representation of its normal form. This is achieved simply by composing reflection with reification. This definition (given formally below) will *not* return the standard representation of the normal form of the term, but rather exploit the fact that every normal form term M can be represented as rep M. This is also a weakness, since the internal structure of M is lost (unlike in the standard representation). In a practical language there seems to be no way around this deficiency. For instance, in a compiled language it is not clear how one could reify the target machine code.

$$\begin{aligned}
&\texttt{reify} &&: \Delta\alpha\,.\,\alpha \to \pi\alpha,\\
&\texttt{reify} &&\equiv \texttt{rep},\\
&\texttt{eval} &&: \Delta\alpha\,.\,\pi\alpha \to \pi\alpha,\\
&\texttt{eval} &&\equiv \Lambda\alpha\,.\,\lambda x{:}\pi\alpha\,.\,\texttt{reify}[\alpha](\texttt{reflect}[\alpha]x).
\end{aligned}$$

Theorem 23 (Correctness of eval). *Let* $\Gamma, \Pi \vdash N \in \pi\alpha$ *be some (not necessarily standard) representation of the term* M. *Then* $\texttt{eval}[\alpha]N$ *is a representation of the normal form of* M.

Proof. Follows directly from Theorem 22. □

We do not have a simple and intuitive characterization of exactly which functions are definable over the given representation of programs. In particular, we do not know whether the apparently simpler one-step outermost β-reduction is representable, but it appears that it is not. The problem is that the first argument to lam expects a function of type $\alpha \to \pi\beta$, not of type $\pi\alpha \to \pi\beta$. Unfortunately, our lack of understanding of exactly what is definable has prevented us from finding more practical programming applications of our metacircularity results.

4.4. Representing inductively defined types and iteration

So far we have been able to interpret F_2 in an enrichment of F_3 that contains some new representation constants and an iteration schema. The purpose of this section is to show that we can eliminate these additional constants: we will explicitly define in pure F_3 a parameterized type π and terms `rep`, `lam`, `app`, `typlam`, and `typapp` to represent programs, and also a term `itprog` that satisfies the reduction property of Definition 21.

The basic problem is to be able to explicitly define a function π from types to types, such that $\pi\alpha$ is a type representing programs of type α. The usual, well-known approach for defining inductive data types in the second-order polymorphic λ-calculus (see [2, 30]) fails (although we do not have a proof that such a representation is impossible). The data types that have been shown to be representable in F_2 either have constructors that are not polymorphic (such as $\mathbf{nat} \equiv \Delta\alpha\,.\,\alpha \to (\alpha \to \alpha) \to \alpha$, which has constructors `zero`:**nat** and `succ`:**nat** → **nat**), or have the property that the type variables in the constructor are uniform over the whole data type (such as $\mathbf{list} \equiv \lambda\alpha\,.\,\Delta\theta\,.\,(\alpha \to \theta \to \theta) \to \theta \to \theta$ with constructors `cons`:$\Delta\theta\,.\,\theta \to \mathbf{list}\ \theta \to \mathbf{list}\ \theta$ and `nil`:$\Delta\theta\,.\,\mathbf{list}\ \theta$). This allows the definitions of the constructors to be uniform over this type variable (see the discussion of uniform parameterization in [27]).

An attempt at a straightforward extension of this approach to the case of a data type of programs fails, since a program of type β may have components of type $\alpha \to \beta$ and α, and thus in fact of arbitrary type. This problem can be dealt with in the third-order polymorphic λ-calculus, since in it one can explicitly use a function from types to types that maps the type of the components to the type of a term.

In analogy to Church's representation of natural numbers in the λ-calculus, each program is represented as its own iteration function. That is, in the end we would like to define (omitting some types of bound variables)

$$\texttt{itprog} \equiv \Lambda\Theta{:}\mathrm{Type} \to \mathrm{Type}\,.\,\lambda h_1 \ldots \lambda h_5\,.\,\Lambda\alpha\,.\,\lambda x{:}\pi\alpha\,.\,x[\Theta]h_1 \ldots h_5\,.$$

From this one can infer what the definition of π will have to be. Each line is annotated with a corresponding constructor function that is defined below.

$$\begin{aligned}
\pi \equiv\ & \lambda\gamma\,.\,\Delta\Theta{:}\mathrm{Type} \to \mathrm{Type}\,. \\
& (\Delta\alpha\,.\,\alpha \to \Theta\alpha) && (*\texttt{rep}*) \\
& \to (\Delta\alpha\Delta\beta\,.\,(\alpha \to \Theta\beta) \to \Theta(\alpha \to \beta)) && (*\texttt{lam}*) \\
& \to (\Delta\alpha\Delta\beta\,.\,\Theta(\alpha \to \beta) \to \Theta\alpha \to \Theta\beta)) && (*\texttt{app}*) \\
& \to (\Delta\alpha{:}\mathrm{Type} \to \mathrm{Type}\,.\,(\Delta\theta\,.\,\Theta(\alpha\theta)) \to \Theta(\Delta\theta\,.\,\alpha\theta)) && (*\texttt{typlam}*) \\
& \to (\Delta\alpha{:}\mathrm{Type} \to \mathrm{Type}\,.\,\Theta(\Delta\theta\,.\,\alpha\theta) \to (\Delta\theta\,.\,\Theta(\alpha\theta))) && (*\texttt{typapp}*) \\
& \to \Theta\gamma.
\end{aligned}$$

This is a special case of a very general transformation from an inductive definition of a data type into an encoding into F_ω described in [27]. The definitions of the constructors in this encoding can be found in Fig. 1.

$$
\begin{array}{lcl}
\texttt{rep} & : & \Delta\alpha\,.\,\alpha \to \pi\alpha \\
\texttt{rep} & \equiv & \Lambda\alpha\,\lambda x{:}\alpha\,. \\
 & & \Lambda\Theta\,\lambda rep\,\lambda lam\,\lambda app\,\lambda typlam\,\lambda typapp\,. \\
 & & rep[\alpha]x
\end{array}
$$

$$
\begin{array}{lcl}
\texttt{lam} & : & \Delta\alpha\Delta\beta\,.\,(\alpha \to \pi\beta) \to \pi(\alpha \to \beta) \\
\texttt{lam} & = & \Lambda\alpha\,\Lambda\beta\,\lambda f{:}\alpha \to \pi\beta\,. \\
 & & \Lambda\Theta\,\lambda rep\,\lambda lam\,\lambda app\,\lambda typlam\,\lambda typapp\,. \\
 & & lam\,[\alpha][\beta](\lambda x{:}\alpha\,.\,f x\,[\Theta]\ rep\ lam\ app\ typlam\ typapp)
\end{array}
$$

$$
\begin{array}{lcl}
\texttt{app} & : & \Delta\alpha\,\Delta\beta\,.\,\pi(\alpha \to \beta) \to \pi\alpha \to \pi\beta \\
\texttt{app} & \equiv & \Lambda\alpha\,\Lambda\beta\,\lambda x{:}\pi(\alpha \to \beta)\,\lambda y{:}\pi\alpha\,. \\
 & & \Lambda\Theta\,\lambda rep\,\lambda lam\,\lambda app\,\lambda typlam\,\lambda typapp\,. \\
 & & app[\alpha][\beta](x[\Theta]\ rep\ lam\ app\ typlam\ typapp) \\
 & & (y[\Theta]\ rep\ lam\ app\ typlam\ typapp)
\end{array}
$$

$$
\begin{array}{lcl}
\texttt{typlam} & : & \Delta\alpha{:}\mathrm{Type} \to \mathrm{Type}\,.\,(\Delta\theta\,.\,\pi(\alpha\theta)) \to \pi(\Delta\theta\,.\,\alpha\theta) \\
\texttt{typlam} & \equiv & \Lambda\alpha{:}\mathrm{Type} \to \mathrm{Type}\,\lambda f{:}\Delta\theta\,.\,\pi(\alpha\theta)\,. \\
 & & \Lambda\Theta\,\lambda rep\,\lambda lam\,\lambda app\,\lambda typlam\,\lambda typapp\,. \\
 & & typlam\,[\alpha](\Lambda\theta\,.\,f[\theta][\Theta]\ rep\ lam\ app\ typlam\ typapp)
\end{array}
$$

$$
\begin{array}{lcl}
\texttt{typapp} & : & \Delta\alpha{:}\mathrm{Type} \to \mathrm{Type}\,.\,\pi(\Delta\theta\,.\,\alpha\theta) \to (\Delta\theta\,.\,\pi(\alpha\theta)) \\
\texttt{typapp} & \equiv & \Lambda\alpha{:}\mathrm{Type} \to \mathrm{Type}\,\lambda f{:}\pi(\Delta\theta\,.\,\alpha\theta)\,\Lambda\theta\,. \\
 & & \Lambda\Theta\,\lambda rep\,\lambda lam\,\lambda app\,\lambda typlam\,\lambda typapp\,. \\
 & & typapp\,[\alpha](f[\Theta]\ rep\ lam\ app\ typlam\ typapp)\,[\theta]
\end{array}
$$

Fig. 1. Definition of program constructors for F_2 in F_3.

We thus can eliminate the context Π^+ and the additional reduction rule for iteration and give a representation of programs in pure F_3.

Theorem 24 (Representation in pure F_3). *Let $\hat{M}$ and $\hat{\alpha}$ be the result of substituting the definitions above for variables π,* rep, lam, app, typlam, typapp, *and* itprog *in a term M or type α, respectively. If $\Gamma, \Pi^+ \vdash M \in \pi\alpha$ then $\Gamma \vdash \hat{M} \in \widehat{\pi\alpha}$. Moreover, if $M =_{\beta\eta\iota} N$ then $\hat{M} =_{\beta\eta} \hat{N}$.*

Proof. This is an instance of the general representation theorem for inductively defined types in [23, 27]. □

The crucial step in the definition of eval is the definition of reflect, which maps the representation of a term of type $\pi\alpha$ into a term of type α, that is,

reflect : $\Delta\alpha\,.\,\pi\alpha \to \alpha$. In order to obtain its definition in pure F_3, we simply match up the general schema of iteration from Definition 19 with the definition of reflect (Definition 20) to obtain expressions for $h_1, \ldots, h_5$. Each h_i turns out to be a variant of the identity function:

$$
\begin{aligned}
\texttt{reflect} \;&:\; \Delta\gamma\,.\,\pi\gamma \to \gamma \\
\texttt{reflect} \;&\equiv\; \texttt{itprog}[\lambda\delta\,.\,\delta](\Lambda\alpha\,.\,\texttt{id}[\alpha])(\Lambda\alpha\Lambda\beta\,.\,\texttt{id}[\alpha \to \beta]) \\
&\qquad (\Lambda\alpha\Lambda\beta\,.\,\texttt{id}[\alpha \to \beta]) \\
&\qquad (\Lambda\alpha\Lambda\beta\,.\,\texttt{id}[\alpha \to \beta]) \\
&\qquad (\Lambda\alpha{:}\text{Type} \to \text{Type}\,.\,\texttt{id}[\Delta\theta\,.\,\alpha\theta]) \\
&\qquad (\Lambda\alpha{:}\text{Type} \to \text{Type}\,.\,\texttt{id}[\Delta\theta\,.\,\alpha\theta]) \\
&\equiv\; \Lambda\gamma\lambda p{:}\pi\gamma\,.\,p[\lambda\delta\,.\,\delta] \\
&\qquad (\Lambda\alpha\,.\,\texttt{id}[\alpha]) \\
&\qquad (\Lambda\alpha\Lambda\beta\,.\,\texttt{id}[\alpha \to \beta]) \\
&\qquad (\Lambda\alpha\Lambda\beta\,.\,\texttt{id}[\alpha \to \beta]) \\
&\qquad (\Lambda\alpha{:}\text{Type} \to \text{Type}\,.\,\texttt{id}[\Delta\theta\,.\,\alpha\theta]) \\
&\qquad (\Lambda\alpha{:}\text{Type} \to \text{Type}\,.\,\texttt{id}[\Delta\theta\,.\,\alpha\theta]).
\end{aligned}
$$

This definition highlights the fact that a program is represented as its own iteration function. The ability of a program to be evaluated is captured in the representation itself—externally we simply supply identity functions.

5. Application to other calculi

Let us first deal with the most obvious question: since F_2 can be reflected in F_3 one might expect that F_3 could be reflected in F_4. However, the construction as given does not extend to this case (as was erroneously claimed in [28]). Can we modify the construction to obtain an interpreter for all of F_ω ? The answer is yes, but we have to modify our construction to add another level in addition to terms, types, and kinds. Perhaps the most uniform way of doing this is to introduce *universes* as in related systems such as the Generalized Calculus of Construction (CC^ω) [6, 16]. This is beyond the scope of this paper, and so we simply give the construction as it would appear if one additional level is added explicitly, thereby allowing an interpreter for F_ω to be written.

The way in which this additional level is added is straightforward for our purposes: we need variables κ ranging over kinds and a way to explicitly abstract terms over kinds.

Definition 25 (*Calculus* F_ω^+). To the syntactic categories of Definition 1 we add

Kinds	K	$::= \ldots \mid \kappa,$
Types	α	$::= \ldots \mid \Delta^+\kappa\,.\,\alpha,$
Terms	M	$::= \ldots \mid \Lambda^+\kappa\,.\,M \mid M[K]^+,$
Contexts	Γ	$::= \ldots \mid \kappa{:}\text{Kind}.$

Λ^+ stands for abstraction over kinds at the level of terms, and a function thus formed has a type of the form $\Delta^+\kappa.\,\alpha$ and can be applied via $[\,]^+$. The inference rules from Section 3 must be modified in the obvious way so that the judgment $\vdash K \in \text{Kind}$ is parameterized by a context, that is, $\Gamma \vdash K \in \text{Kind}$. We also add the following new deduction rules:

$$\frac{\vdash \Gamma \text{ context} \qquad \kappa{:}\text{Kind in } \Gamma}{\Gamma \vdash \kappa \in \text{Kind}}$$

$$\frac{\Gamma, \kappa{:}\text{Kind} \vdash \alpha \in \text{Type}}{\Gamma \vdash \Delta^+\kappa.\,\alpha \in \text{Type}}$$

$$\frac{\Gamma \text{ context}}{\Gamma, \kappa{:}\text{Kind context}}$$

$$\frac{\Gamma, \kappa{:}\text{Kind} \vdash M \in \alpha}{\Gamma \vdash \Lambda^+\kappa.\,M \in \Delta^+\kappa.\,\alpha}$$

$$\frac{\Gamma \vdash M \in \Delta^+\kappa.\,\alpha \qquad \Gamma \vdash K \in \text{Kind}}{\Gamma \vdash M[K]^+ \in [K/\kappa]\alpha}$$

All of the desirable properties such as strong normalization and decidability of type-checking of F_ω are preserved in F_ω^+ (see, for example, [19] for the proofs in a much stronger system of which F_ω^+ is only a small fragment).

The next step is to modify the construction in Sections 4 and 4.4. The crucial change is in the definition of π: all the other changes follow almost automatically. Consider

$$\texttt{typlam} : \Delta\alpha{:}\text{Type} \to \text{Type}.\,(\Delta\theta.\,\pi(\alpha\theta)) \to \pi(\Delta\theta.\,\alpha\theta).$$

This must now be generalized, since abstractions in F_ω may also range over variables of kind Type → Type, Type → Type → Type, and so on. In order to represent *all* of these in a uniform way, we need a family of constructors, indexed by a kind K:

$$\texttt{typlam}_K : \Delta\alpha{:}K \to \text{Type}.\,(\Delta\theta{:}K.\,\pi(\alpha\theta)) \to \pi(\Delta\theta{:}K.\,\alpha\theta).$$

In [26] we proposed using global definitions and definitional equality to solve this problem, here we add a way of explicitly abstracting over kinds. In our notation from above, typlam will then have the type

$$\texttt{typlam} : \Delta^+\kappa.\,\Delta\alpha{:}\kappa \to \text{Type}.\,(\Delta\theta{:}\kappa.\,\pi(\alpha\theta)) \to \pi(\Delta\theta{:}\kappa.\,(\alpha\theta)).$$

The modified standard representation function then reads as follows:

$$
\begin{array}{ll}
\text{If } x \in \alpha & \text{then } \bar{x} = \mathtt{rep}[\alpha]x. \\
\text{If } \lambda x{:}\alpha\,.\, M \in \alpha \to \beta & \text{then } \overline{\lambda x{:}\alpha\,.\, M} = \mathtt{lam}[\alpha][\beta](\lambda x{:}\alpha\,.\, \bar{M}). \\
\text{If } M \in \alpha \to \beta \text{ and } N \in \alpha & \text{then } \overline{M\,N} = \mathtt{app}[\alpha][\beta]\bar{M}\,\bar{N}. \\
\text{If } \Lambda\theta{:}K\,.\, M \in \Delta\theta{:}K\,.\,\alpha & \text{then } \overline{\Lambda\theta{:}K\,.\, M} \\
 & \quad = \mathtt{typlam}[K]^{+}[\lambda\theta{:}K\,.\,\alpha](\Lambda\theta{:}K\,.\,\bar{M}). \\
\text{If } M \in \Delta\theta{:}K\,.\,\alpha & \text{then } \overline{M[\beta]} = \mathtt{typapp}[K]^{+}[\lambda\theta{:}K\,.\,\alpha]\bar{M}[\beta].
\end{array}
$$

The type of typapp has to be changed in a way analogous to typlam leading to the following definition of π generalized from Section 4.4:

$$
\begin{array}{ll}
\pi \equiv \lambda\gamma\,.\,\Delta\Theta{:}\mathrm{Type} \to \mathrm{Type}\,. & \\
\quad (\Delta\alpha\,.\,\alpha \to \Theta\alpha) & (*\mathtt{rep}*) \\
\quad \to (\Delta\alpha\Delta\beta\,.\,(\alpha \to \Theta\beta) \to \Theta(\alpha \to \beta)) & (*\mathtt{lam}*) \\
\quad \to (\Delta\alpha\Delta\beta\,.\,\Theta(\alpha \to \beta) \to \Theta\alpha \to \Theta\beta)) & (*\mathtt{app}*) \\
\quad \to (\Delta^{+}\kappa\,.\,\Delta\alpha{:}\kappa \to \mathrm{Type}\,.\,(\Delta\theta{:}\kappa\,.\,\Theta(\alpha\theta)) & \\
\qquad \to \Theta(\Delta\theta{:}\kappa\,.\,\alpha\theta)) & (*\mathtt{typlam}*) \\
\quad \to (\Delta^{+}\kappa\,.\,\Delta\alpha{:}\kappa \to \mathrm{Type}\,.\,\Theta(\Delta\theta{:}\kappa\,.\,\alpha\theta) & \\
\qquad \to (\Delta\theta{:}\kappa\,.\,\Theta(\alpha\theta))) & (*\mathtt{typapp}*) \\
\quad \to \Theta\gamma. &
\end{array}
$$

Most of the other definitions of Section 4.4 go through as given, with some changes in the types (which were omitted in Fig. 1). As an example we consider typlam.

$$
\begin{array}{lll}
\mathtt{typlam} & : & \Delta^{+}\kappa\,.\,\Delta\alpha{:}\kappa \to \mathrm{Type}\,.\,(\Delta\theta{:}\kappa\,.\,\pi(\alpha\theta)) \to \pi(\Delta\theta{:}\kappa\,.\,\alpha\theta) \\
\mathtt{typlam} & \equiv & \Lambda^{+}\kappa\,.\,\Lambda\alpha{:}\kappa \to \mathrm{Type}\;\lambda f{:}\Delta\theta{:}\kappa\,.\,\pi(\alpha\theta)\,. \\
 & & \Lambda\Theta\;\lambda \mathit{rep}\;\lambda \mathit{lam}\;\lambda \mathit{app}\;\lambda \mathit{typlam}\;\lambda \mathit{typapp}\,. \\
 & & \mathit{typlam}\;[\kappa]^{+}[\alpha](\Lambda\theta{:}\kappa\,.\,f[\theta][\Theta]\;\mathit{rep}\;\mathit{lam}\;\mathit{app}\;\mathit{typlam}\;\mathit{typapp})
\end{array}
$$

For the definition of reflect we get

$$
\begin{array}{lll}
\mathtt{reflect} & : & \Delta\gamma\,.\,\pi\gamma \to \gamma \\
\mathtt{reflect} & \equiv & \Lambda\gamma\lambda p{:}\pi\gamma\,.\,p[\lambda\delta\,.\,\delta] \\
 & & \quad (\Lambda\alpha\,.\,\mathtt{id}[\alpha]) \\
 & & \quad (\Lambda\alpha\,\Lambda\beta\,.\,\mathtt{id}[\alpha \to \beta]) \\
 & & \quad (\Lambda\alpha\,\Lambda\beta\,.\,\mathtt{id}[\alpha \to \beta]) \\
 & & \quad (\Lambda^{+}\kappa\,.\,\Lambda\alpha{:}\kappa \to \mathrm{Type}\,.\,\mathtt{id}[\Delta\theta{:}\kappa\,.\,\alpha\theta]) \\
 & & \quad (\Lambda^{+}\kappa\,.\,\Lambda\alpha{:}\kappa \to \mathrm{Type}\,.\,\mathtt{id}[\Delta\theta{:}\kappa\,.\,\alpha\theta]).
\end{array}
$$

The representation theorems go through in the same way as before, but now any term in F_ω can be represented and evaluated. Even though the uniform representation and definition of the evaluation function is in F_ω^{+}, evaluation of a given term in F_n "takes place" in F_{n+1}, since any given term in F_n will only use finitely many kinds.

6. Conclusions

We conclude that metacircularity is very nearly attainable in a statically typed language. Unfortunately, this does not seem to imply that the same language is also suitable for *typed metaprogramming*: the construction of statically typed programs (called *metaprograms*) that construct, analyze, and manipulate other programs (which are called the *object programs*). It is this problem which provided the original motivation for the construction presented in this paper.

With regard to typed metaprogramming, it seems that we have little to add to what is already known, despite the fact that we have developed a simple extension to F_ω that allows all of F_ω to be represented. Our experience has been that evaluation is just about the only useful function definable over this representation. Other interesting metaprogramming tasks, such as partial evaluation, macro expansion, program transformations, and so on, do not seem to be expressible.

The precise reasons for these difficulties have eluded us thus far, and as a result we have yet to prove any negative results. However, there are a number of plausible explanations which center on the issue of how to model abstraction.

- If one models abstraction in the object language by abstraction in the meta-language (as we have done here), then static typing does not seem to be a major obstacle to useful metaprograms. Instead, the problem seems to be an insufficient degree of access to the intensional structure of programs. In a functional language, a possible way out may be to preserve intensionality with new language constructs that are parallel to, but separate from extensional function constructors.
- If abstraction is not modeled by abstraction, then static typing becomes a major obstacle to metacircularity. Of course, removing the static typing requirement allows many useful metaprograms to be expressed, as exemplified by Lisp. In a statically typed setting, however, proofs of well-typedness would have to be carried out at the meta-level and, moreover, reflection and reification functions could not be made internal. Still, this approach is promising and has been explored by Howe in the framework of NuPrl [17, 18].

In related research we have been working on the design and implementation of a practical, explicitly polymorphic language along the lines of ML which we call LEAP.[1] For a core of LEAP which encompasses most language fragments described in this paper, we have built a prototype implementation, written largely in the language λProlog [22], and the examples in this paper have been run on our implementation. Two features of our language of importance to the ultimate practicality of LEAP are type reconstruction and type-argument synthesis.

With regard to type reconstruction, we employ the convention of allowing the programmer to omit type information, but with the requirement that "placeholders" be used to mark all applications of functions to types. So, for example, the representation of the polymorphic identity function from Example 13 could be

[1] LEAP is an acronym for a Language with Eval And Polymorphism.

written as

$$\overline{\texttt{id}} = \texttt{typlam[ ]}(\Lambda\alpha\,.\,\texttt{lam[ ][ ]}(\lambda x{:}\alpha\,.\,\texttt{rep[ ]}x)).$$

Though undecidable, we have found in practice that the semi-decision procedure given in [24] for this type reconstruction problem behaves acceptably well. There is much yet to be explored here, however, especially in the practical engineering issues, such as the efficiency of the reconstruction mechanism, its behavior on errors and failures, and the incorporation of a notion of modules.

Even with type reconstruction, we find the requirement of placeholders to be cumbersome. Hence, a mechanism for synthesizing type-argument applications is necessary. This has been noted by others as well, and various methods have been proposed for carrying out this synthesis [9, 29]. In LEAP, we have taken a purely syntactic approach involving the annotation of identifiers in their defining occurrences by the number of type arguments to be inferred at each occurrence of that identifier, thus separating issues of type reconstruction from the issues of argument synthesis. Modifying the left-hand side of the definition of `typlam` and related functions by annotating them with *'s (for example, `typlam*` = ..., and `lam**` = ...), we express the representation of the polymorphic identity as

$$\overline{\texttt{id}} = \overline{\texttt{typlam}}(\Lambda\alpha\,.\,\texttt{lam}(\lambda x{:}\alpha\,.\,\texttt{rep}\,x)).$$

With type reconstruction and type-argument synthesis, as well as inductive type definitions, we obtain a useful and syntactically tractable LEAP language. Of course, lacking a full implementation we can only speculate on the question of its ultimate practicality. However, almost any argument that might be made for ML as a metalanguage can also be made for LEAP. In addition, LEAP is able to represent and manipulate in a type-safe way data with richer type structures than is possible in ML. Just how useful this added power is in practice will require much further investigation and experience.

Other issues to be studied further include the exact extent of the language, in particular with respect to additions such as general recursion, references, exceptions, and so on. We have done some preliminary work along these lines, and have some evidence that such extensions will not destroy the "reflective" properties of LEAP. Another issue is the efficiennt implementation of LEAP. Our efforts here have been directed towards devising efficient implementation strategies for inductively defined data types and recursive functions defined over such types.

We hope to have more to report as the design and implementation of a full LEAP language proceeds.

Acknowledgment

The authors would like to thank Christine Paulin-Mohring for valuable criticism of an earlier draft and also Ken Cline, Scott Dietzen, Jean Gallier, Robert Harper,

Spiro Michaylov, and Benjamin Pierce for many helpful discussions about F_ω, reflection, and metaprogramming.

References

[1] K. Bowen and R. Kowalski, Amalgamating language and metalanguage in logic programming, in: K.L. Clark and S.-A. Tärnlund, eds., *Logic Programming* (Academic Press, New York, 1982) 153–172.

[2] C. Böhm and A. Berarducci, Automatic synthesis of typed Λ-programs on term algebras, *Theoret. Comput. Sci.* **39** (1985) 135–154.

[3] A. Church, *The Calculi of Lambda-Conversion* (Princeton University Press, Princeton, NJ, 1941).

[4] A. Church, A formulation of the simple theory of types, *J. Symbolic Logic* **5** (1940) 56–68.

[5] P. Cointe, Metaclasses are first class: the ObjVlisp model, in: N. Meyrowitz, ed., *OOPSLA '87: Proc. 1987 Conf. on Object-Oriented Programming Systems, Languages and Applications*, Orlando (ACM Press, 1987) 156–167.

[6] T. Coquand, An analysis of Girard's paradox, in: *Symp. on Logic Computer Science*, (IEEE, New York, 1986) 227–236.

[7] T. Coquand and G. Huet, The Calculus of Constructions, *Inform. and Comput.* **76**(2/3) (1988) 95–120.

[8] T. Coquand and C. Pauling-Mohring, Inductively defined types, Talk presented at the Workshop on Programming Logic, University of Göteborg and Chalmers University of Technology, 1989.

[9] Project Formel, The Calculus of Constructions, INRIA-ENS, Documentation and User's Guide, Version 4.10, 1989.

[10] D.P. Friedman and M. Wand, Reification: reflection without metaphysics, in: *Proc. 1984 ACM Symp. on Lisp and Functional Programming* (ACM Press, 1984) 348–355.

[11] J.H. Gallier, On Girard's "Candidats de Réductibilité; in: P. Odifreddi, ed., *Logic and Computer Science* (Academic Press, New York, 1990).

[12] J.-Y. Girard, Interprétation fonctionelle et élimination des coupures dee l'arithmétique d'ordre supérieur, Ph.D. Thesis, Université Paris VII, 1972.

[13] J.-Y. Girard, Une extension de l'interprétation de Gödel à l'analyse, et son application a l'élimination des coupures dans l'analyse et la théorie des types, in: J.E. Fenstad, ed., *Proc. 2nd Scandinavian Logic Symp.* (North-Holland, Amsterdam, 1971) 63–92.

[14] J.-Y. Girard, Y. Lafont, and P. Taylor, *Proofs and Types*, Cambridge Tracts in Theoretical Computer Science **7** (Cambridge University Press, Cambridge, 1989).

[15] R. Harper, F. Honsell and G. Plotkin, A framework for defining logics, *J. ACM* (to appear); a preliminary version appeared in: *Symp. on Logic in Computer Science* (1987) 194–204.

[16] R. Harper and R. Pollack, Type checking, universe polymorphism, and typical ambiguity in the Calculus of Constructions, in: *TAPSOFT '89, Proc. Internat. Joint Conf. on Theory and Practice in Software Development*, Barcelona, Spain, Lecture Notes in Computer Science **352** (Springer, Berlin, 1989) 241–256.

[17] D.J. Howe, Automating Reasoning in an implementation of Constructive type theory. Ph.D. Thesis, Computer Science Department, Cornell University, 1987.

[18] D.J. Howe, Computational metatheory in Nuprl, in: E. Lusk and R. Overbeek, eds., *9th Internat. Conf. on Automated Deduction*, Argonne, Illinois, Lecture Notes in Computer Science **310** (Springer, Berlin, 1988) 238–257.

[19] Z. Luo, ECC, an extended Calculus of Constructions, in: *4th Ann. Symp. on Logic in Computer Science* (IEEE Computer Society Press, 1989) 386–395.

[20] J. McCarthy, Recursive functions of symbolic expressions and their computation by machine, *Comm. ACM* **3**(4) (1960) 184–195.

[21] R. Milner, The Standard ML core language, *Polymorphism* **II**(2) (1985); also Technical Report ECS-LFCS-86-2, University of Edinburgh, Edinburgh, Scotland, 1986.

[22] G. Nadathur and D. Miller, An overview of λProlog, in: R.A. Kowalski and K.A. Bowen, eds., *Logic Programming: Proc. 5th Internat. Conf. Symp.*, Volume 1 (MIT Press, Cambridge, MA, 1988) 810–827.

[23] C. Paulin-Mohring, Extraction de programmes dans le Calcul des Constructions, Ph.D. Thesis, Université Paris VII, 1989.

[24] F. Pfenning, Partial polymorphic type inference and higher-order unification, in: *Proc. 1988 ACM Conf. on Lisp and Functional Programming*, Snowbird, Utah (ACM Press, 1988) 153–163; also available as Ergo Report 88-048, School of Computer Science, Carnegie Mellon University, Pittsburgh.

[25] F. Pfenning and C. Elliott, Higher-order abstract syntax, in: *Proc. SIGPLAN '88 Symp. on Language Design and Implementation*, Atlanta, Georgia (ACM Press, 1988) 199–208; available as Ergo Report 88-036, School of Computer Science, Carnegie Mellon University, Pittsburgh.

[26] F. Pfenning and P. Lee, LEAP: a language with eval and polymorphism, in: *TAPSOFT '89, Proc. Internat. Joint Conf. on Theory and Practice in Software Development*, Barcelona, Spain, Lecture Notes in Computer Science **352** (Springer, Berlin, 1989) 345–359; also available as Ergo Report 88-065, School of Computer Science, Carnegie Mellon University, Pittsburgh.

[27] F. Pfenning and C. Paulin-Mohring, Inductively defined types in the Calculus of Constructions, in: *Proc. 5th Conf. on the Mathematical Foundations of Programming Semantics*, Lecture Notes in Computer Science (Springer, Berlin, 1989); available as Ergo Report 88-069, School of Computer Science, Carnegie Mellon University, Pittsburgh.

[28] B. Pierce, S. Dietzen and S. Michaylov, Programming in higher-order typed lambda-calculi, Technical Report CMU-CS-89-111, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1989.

[29] R. Pollack, The theory of LEGO, Unpublished manuscript and documentation, 1988.

[30] J. Reynolds, Three approaches to type structure, in: H. Ehrig, C. Floyd, M. Nivat and J. Thatcher, eds., *Mathematical Foundations of Software Development*, Lecture Notes in Computer Science **185** (Springer, Berlin, 1985) 97–138.

[31] J. Reynolds, Towards a theory of type structure, in; *Proc. Coll. Programmation*, Lecture Notes in Computer Science **19** (Springer, Berlin, 1974) 408–425.

[32] J. Reynolds, Definitional interpreters for higher-order programming languages, in: *Proc. 25th ACM National Conf.* (ACM, New York, 1972) 717–740.

[33] B.C. Smith, Reflection and semantics in a procedural language, Technical Report MIT-LCS-TR-272, Massachusetts Institute of Technology, Cambridge, MA, 1982.

[34] B.C. Smith, Reflection and semantics in Lisp, in: *Proc 11th ACM Symp. on Principles of Programming Languages*, Salt Lake City (ACM, 1984) 23–35.

[35] G.L. Steele and G.J. Sussman, The revised report on SCHEME—a dialect of LISP, AI Memo 452, MIT, Cambridge, 1978.

[36] M.D. Wand and D.P. Friedman, The mystery of the tower revealed: a nonreflective description of the reflective tower, *Lisp Symbolic Comput.* **1**(1) (1988) 11–38.